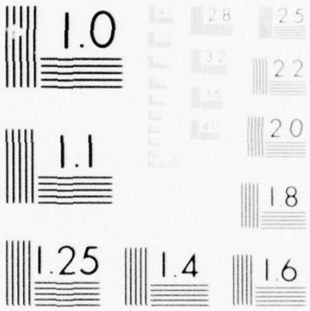
AD-A046 458 HONEYWELL INC MINNEAPOLIS MINN SYSTEMS AND RESEARCH --ETC F/G 9/2
A GRAPH THEORETIC APPROACH TO FAULT TOLERANT COMPUTING.(U)
SEP 77 W L HEIMERDINGER, Y W HAN F44620-75-C-0053
UNCLASSIFIED F0357 AFOSR-TR-77-1255 NL

| OF |

AD
A046458

END
DATE
FILMED
12-77
DDC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

See 1473

3

A GRAPH THEORETIC APPROACH TO FAULT TOLERANT COMPUTING

Final Report

SEPTEMBER 12, 1977

W. L. Heimerdinger, Y. W. Han

AIR FORCE OFFICE OF
SCIENTIFIC RESEARCH
(AFSC)

Bolling Air Force Base, D.C. 20332

Contract No. F44620-75-C-0053

Honeywell

A GRAPH THEORETIC APPROACH TO FAULT TOLERANT COMPUTING

Final Report

September 12, 1977

Contract No. F44620-75-C-0053

Systems & Research Center
2700 RIDGWAY PARKWAY
MINNEAPOLIS, MINNESOTA 55413

Printed in U.S.A.

TABLE OF CONTENTS

SECTION I

INTRODUCTION AND SUMMARY

INTRODUCTION

This report documents the activities in the second year of a two year investigation of a graph theoretic approach to fault tolerance for the Air Force Office of Scientific Research. This is part of a continuing effort also sponsored by the Office of Naval Research and by Honeywell, Inc. to develop a unified approach to the analysis of fault tolerant digital systems based on graph theory. Earlier efforts have examined existing graphical models and found a number of them to be suitable for fault tolrance modeling. Two models, Petri Nets and LOGOS were found to be particularly suitable. A subsequent effort examined available results in Petri net theory for properties and relationships applicable to fault tolerance phenomena.

Previous investigations have concentrated on the sequencing and control aspects of systems and did not explicitly consider time as a parameter in the model. The effort documented here focused on the incorporation of data aspects of the system in the model and on an explicit representation of time in the model.

In Section II, the work on data representation considers data modeling approaches that use special value tokens to trace the flow of data in a single graph and two-graph modeling approaches in which data interactions are represented in a separate data graph. We conclude that the two-graph approach is more appealing for fault tolerance analysis because of its ability to more closely approximate the data structure at lower levels in a system.

We have concluded that an effective approach to fault tolerant system analysis must begin at the functional level of system description. At this level, it is important to be able to model software structures such as arrays, lists, and queues in a natural way. At the next lower level of abstraction we become concerned with software intercommunication mechanisms, including procedure calls and parameter passing schemes. Thus, we conducted a number of modeling exercises in which we use the two-graph model to represent variables with hierarchically nested scope, procedure calls, and a variety of parameter passing mechanisms.

In Section III of this report, we describe a procedure for applying a data flow analysis procedure originally developed for compiler optimization to the tracing of data contamination in a fault tolerant system. We describe a transformation scheme to convert our labeled graph model to a direcιed graph representation compatible with data flow analysis algorithms and a number of other directed graph results.

In Section IV, we discuss the inclusion of time in a labeled graph fault tolerance model. This recognizes the fact that a system which sequences correctly but does not complete critical sequences within time deadlines must be considered faulty in a real time environment. We conclude that we may modify a labeled graph to associate two real numbers with each transition or operator to denote upper and lower bounds on the firing time of the transition or operation time of the operator. This provides us with a model having greater representation capability than a simple Petri net.

SUMMARY

The above results have convinced us that a two-graph labeled graph model that associates two time parameters with each transition or operation is a feasible and effective method of representing a fault tolerant digital system for analysis purposes. We have not specified the exact syntax of a single model in this effort, but we feel such a definition can be made using these results in a straightforward way. Additional work is needed to identify data attributes critical to fault tolerance and to include them in the model. This is the subject of an ongoing effort.

We conclude that the two-graph labeled graph approach to fault tolerance modeling is viable and promising.

SECTION II

DATA REPRESENTATION

BACKGROUND

The majority of digital systems amenable to fault tolerance analysis perform sequences of operations on stored data. For analysis purposes, we separate the control (sequencing and coordination) and data (retention and manipulation of stored data) aspects of these systems. A number of labelled graph models have been developed to represent coordination and sequencing activities, but few existing models include both control and data information in the same model.

The earlier tasks in our study of graphical approaches to the analysis of fault tolerant systems have concentrated on the control aspects of systems. This years' effort began the development of a data representation for the fault tolerance model. The goal is the creation of a data representation that allows examination of three issues involved in the manipulation and storage of a data item:

1) Data value - the current value of a data element

2) Data access - the mechanisms used to link the reference to a data item by a programmer to the actual image of the data element as stored in the system

3) Data transformation - the operations allowed on the given data element.

In this section, we first review the relative advantages of a single graph and a two graph approach to data representation. We conclude that both are adequate and that the two graph approach provides better insight into data interactions.

We then use the two graph approach to verify that common data constructs can be effectively represented.

SEPARATE VS COMBINED DATA GRAPHS

There are two major approaches for combining control and data information into a single graphical model. The two graph approach augments a Petri Net-like control graph representing control in the system with a second graph portraying the connectivity of data items and data operations in the system. In this scheme, data operations are sequenced by pairing operators in the data graph with operators in the associated control graph.

The single graph approach essentially interleaves a control graph with a data graph in a single graph which uses two types of links to carry information. Control links carry control tokens to sequence operations in the model as in the two graph model, while data links carry special data tokens to represent data values.

The first activity in this study of data representation compared the single graph and two graph approaches. Two existing labelled graph models were used as the basis for the comparison. The Data Flow Procedure Language developed at MIT was used as the prototype single graph model, and LOGOS, one of the models used earlier in this study was used as the prototype two graph model. A brief summary of the features of Data Flow Procedure

Language follows. LOGOS is summarized in our previous report [36]. The two models are defined in detail in [31] and [19].

DATA FLOW PROCEDURE LANGUAGE FEATURES

Data Flow Procedure Language (DFP) models, like Petri nets, are graphical models in which tokens circúlate to indicate the state of the system. In order to represent data driven events, DFP defines data dependent actors. These include operators, T-gates, F-gates, Boolean actors (AND, NOT, and OR), and decider and merge operators as shown in Figures 1a & b. Actors are interconnected with links, represented by one or more connected edges in the graph. A link acts like a place in a Petri net or a counter cell in a LOGOS model to hold a token. Each edge in a link can contain either one token or no token, just as a place in a safe Petri net can contain at most one token. Tokens in data flow procedures are classified as data tokens or control tokens. Data tokens denote operands of functions (actors and links) and control tokens are used to determine control sequences.

In contrast to Petri net tokens, DFP tokens always have a value; control tokens assume the values TRUE or FALSE and data tokens assume values allowed for the data item being represented. Links correspond to the type of token they carry; control links, represented by arcs with open arrowheads, carry only control tokens, while data links, represented by arcs with solid arrowheads, carry data tokens.

Except for control actors (T-gate, F-gate and merge), a link or an actor is *enabled* when tokens are present on input arcs (edges) and no token is on any output arc. The firing of the actor or the link absorbs the tokens from the input arcs and places tokens on the output arcs.

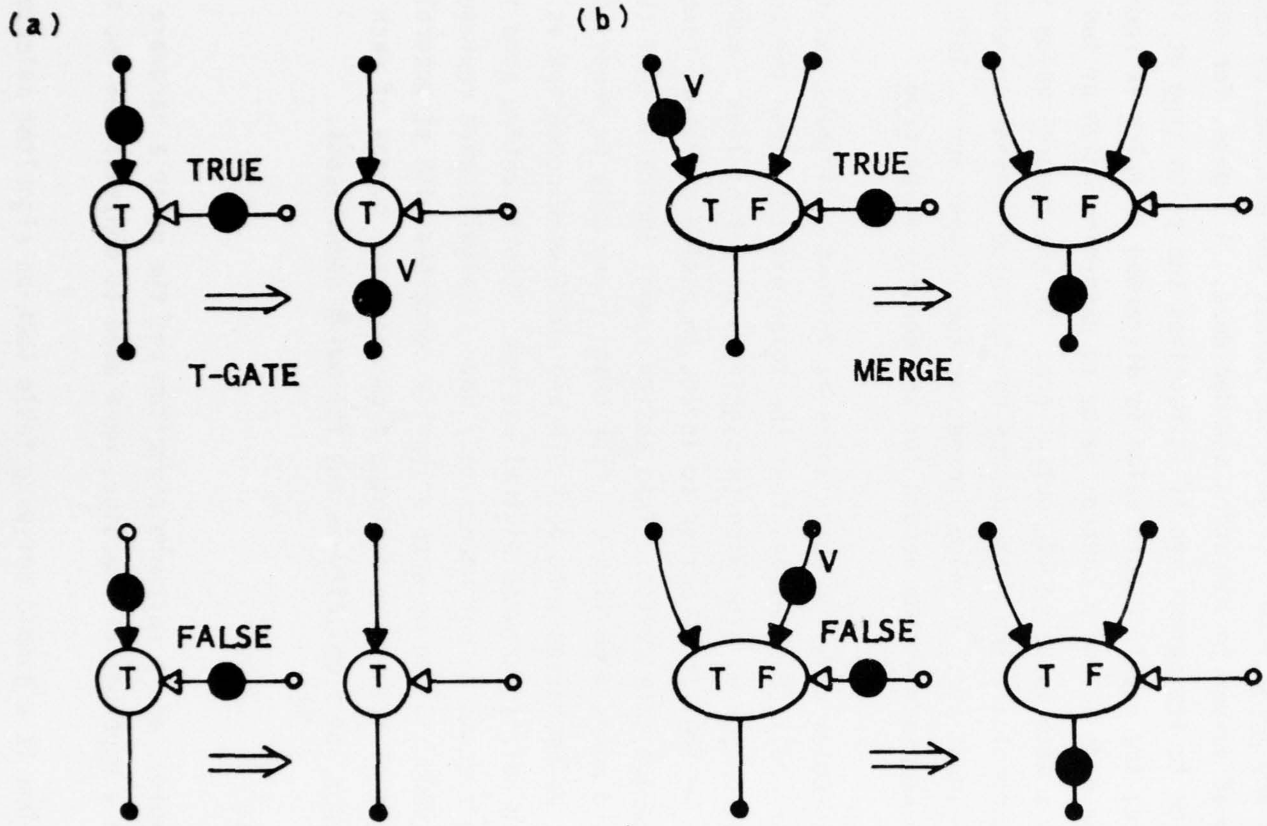


Figure 1.

The gate and merge actors (Figure 1b) permit the outcomes of tests to affect the flow of values to operators and deciders. A T-gate, for example, passes a value on to its output arc if it receives the value true at its control input arc; the received data value is discarded if false is received. The The merge node allows a control value to determine which of two sources supplies a data value to its output arc. If the control value false arrives at the control arc the merge passes on the value present or next to arrive at the F-input arc. A value present at the T-input arc is left undisturbed. The complementary action occurs for the control value true.

Figure 2denotes a switch which works as follows: If there exists a token in the square cell (E signifies that the token exists), then the token on link b passes to link d; otherwise (N signifies no token) link c tokens pass to link d. We feel it is better to invent an assign operator, denoted in Figure 3, to replace this switch. This assign symbol indicates that the initial value of d comes from link b; after that it can only be reassigned through link c. We emphasize that both DFP and LOGOS were conceived as design tools to create well structured digital systems. Consequently, some features of existing systems do not necessarily have straightforward representations in DFP or LOGOS. Since we were primarily concerned with structural and philosophical issues, we deviated from the exact syntax of each model for convenience and simplicity in the following comparisons.

EXAMPLES:

Two examples, one a software algorithm and the other a hardware example based on a hypothetical machine, were used to contrast the two models.

Example One is a simple software table look-up algorithm selected to create a situation in which the value of a data item is used to access the next data item.

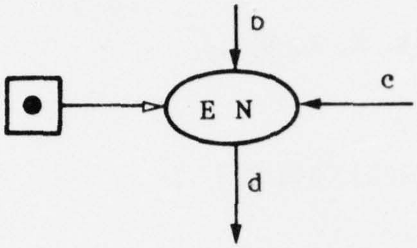
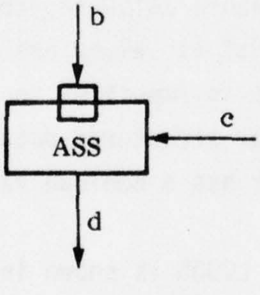


Figure 2.



ASS denotes an Assignment Operation.

Figure 3.

Example One:

```
        PROCEDURE  EXAMPLE (M, N, K, W) ;
        INTEGER:  M, N, W ;
        INTEGER CONSTANT:   K  ;
        INTEGER ARRAY:   A[0:23], B[0:23]  ;
        BEGIN
                 WHILE   N ≠ K DO
                     BEGIN
                         M←B(M);
                         N←A(M);
                     END;
                 W←A(M);
        END;
```

Figure 4 shows a DFP representation of Example One. Note that ARRAY and CONSTANT work as a memory cell; i.e., although we can envision a token being moved out as the related operators activate, there is another token with the same value to replenish it (just as values are rewritten in core memory cells after a destructive read). It is important to note that a token on a data link has a value (or values for structured data) associated with it and that a token on a control link has a Boolean value: true or false.

The same example expressed in LOGOS is shown in Figures 5a and 5b. It consists of two parts: the data graph contains the information on data access and data transformation, and the control graph depicts the control flow of the modeled system.

Example Two is a hardware example based on a hypothetical machine shown in Figures 6a and 6b. This example executes an ADD instruction. Details of the steps executed are listed below.

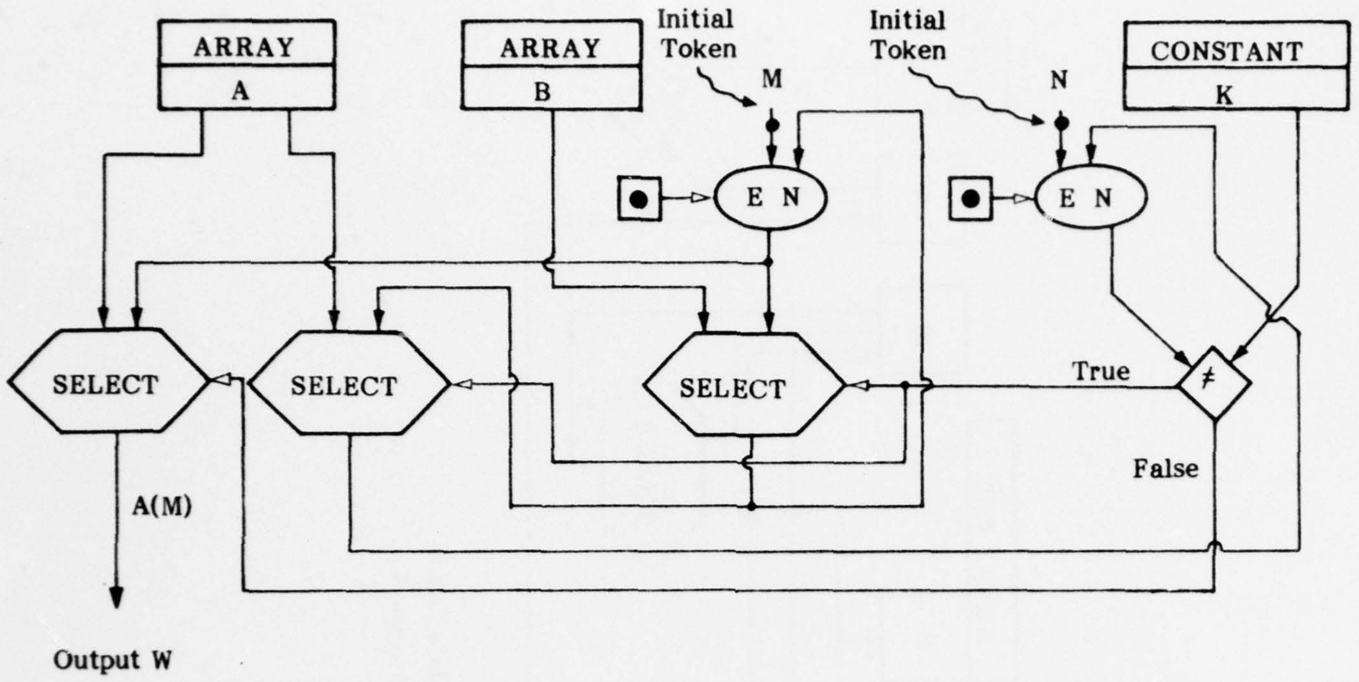
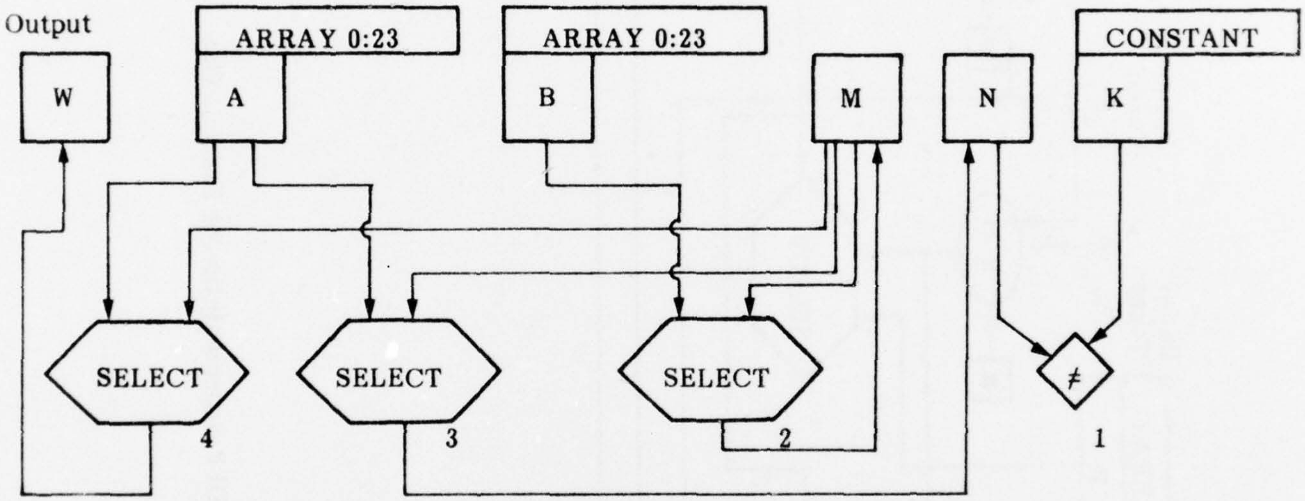
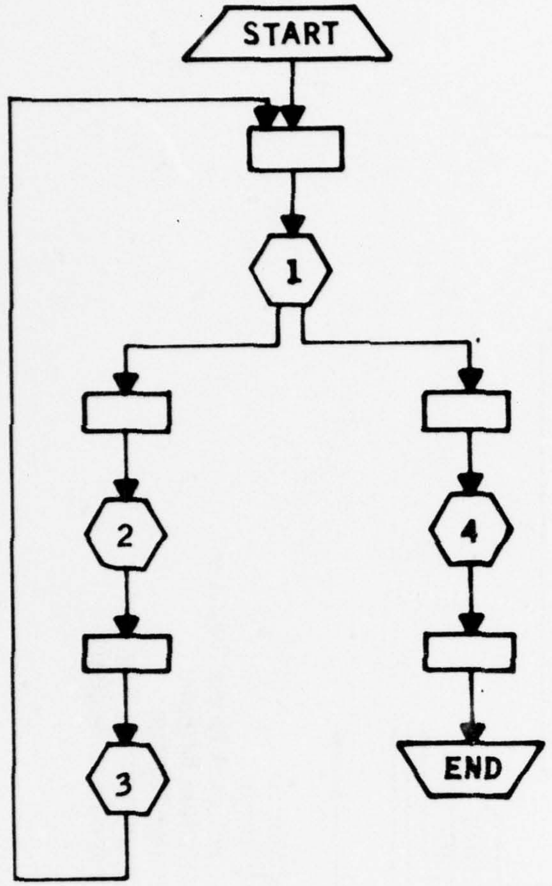


Figure 4 DFP Representation of Example One



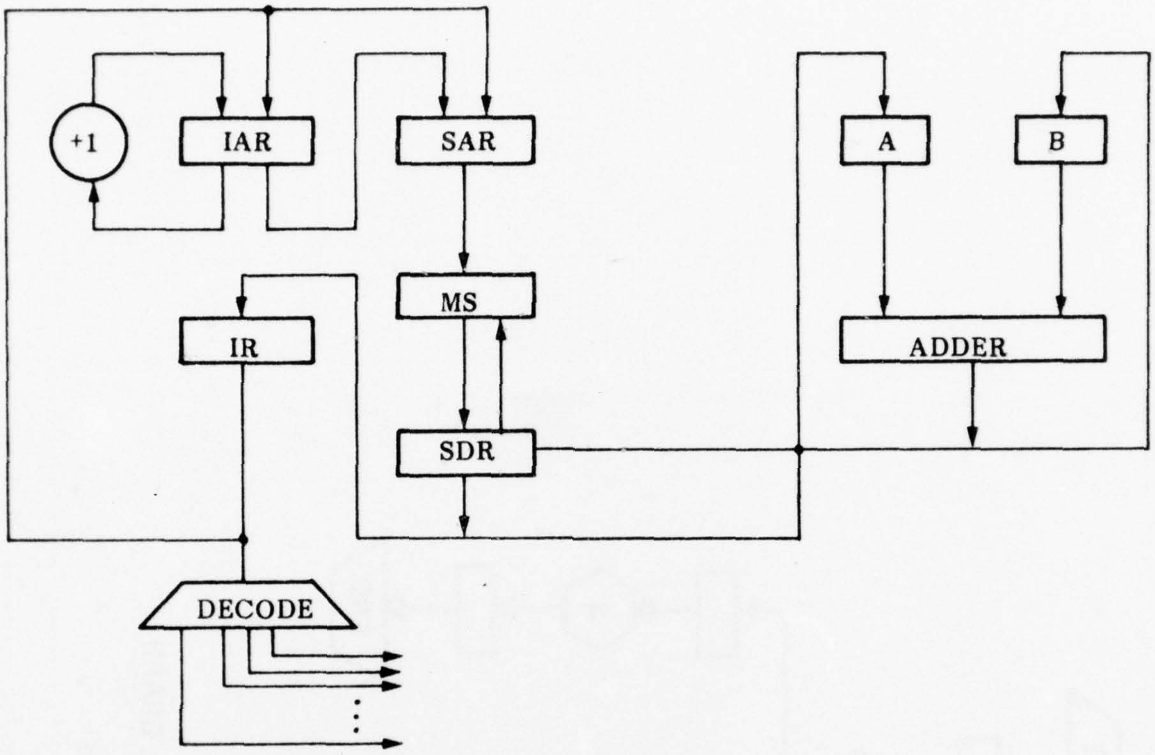
Data Graph
Figure 5a.



(b) CONTROL GRAPH

Figure 5

(a) Hypothetical Machine Structure



A: Register A
B: Register B
IAR: Instruction Address Register
IR: Instruction Register
MS: Memory Storage
SAR: Storage Address Register
SDR: Storage Data Register

(b) Instruction Format

OP Code	Address of Operand One	Address of Operand Two

Figure 6. Hypothetical Machine

Example Two:

IAR → SAR
IAR+1 → IAR
MS [SAR] → IR
Decode OP Code (ADD)
IR [Addr. of Operand 2] → SAR
MS [SAR] → SDR
SDR → B
IR [Addr. of Operand 1] → SAR
MS [SAR] → SDR
SDR → A
A + B → A
A → SDR
SDR → MS [SAR]

Figure 7 illustrates DFP representation of Example Two. We deviated from the original DFP notation by using sequence links, denoted as ---▷ , to specify the precedence relationships between related operators. The operator at the open arrow end of a sequence link cannot start until the other operator completes its operation. Sequence links will be discussed in greater detail later when the construction of a modified single graph model is discussed. The representation of Example Two in LOGOS is shown in Figures 8a and 8b.

COMPARISON OF SINGLE VS DOUBLE GRAPH MODELS

In comparing the two models, we were most interested in the relationship between the data and control portions of the models, the degree to which the models follow the actual logical structure of the system, and the ability of the models to expose parallelism and concurrency in systems. Eight aspects of the models were specifically considered:

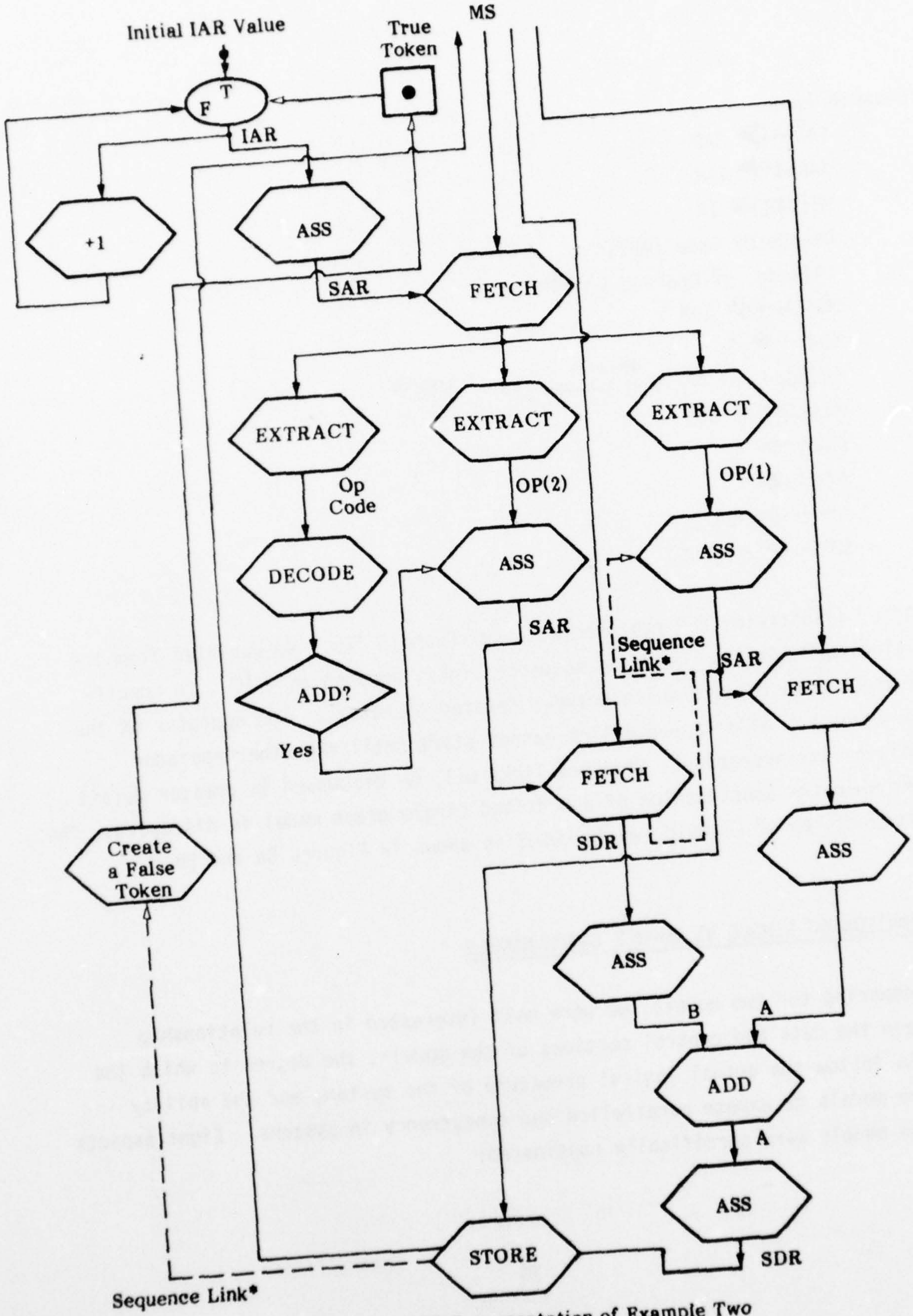
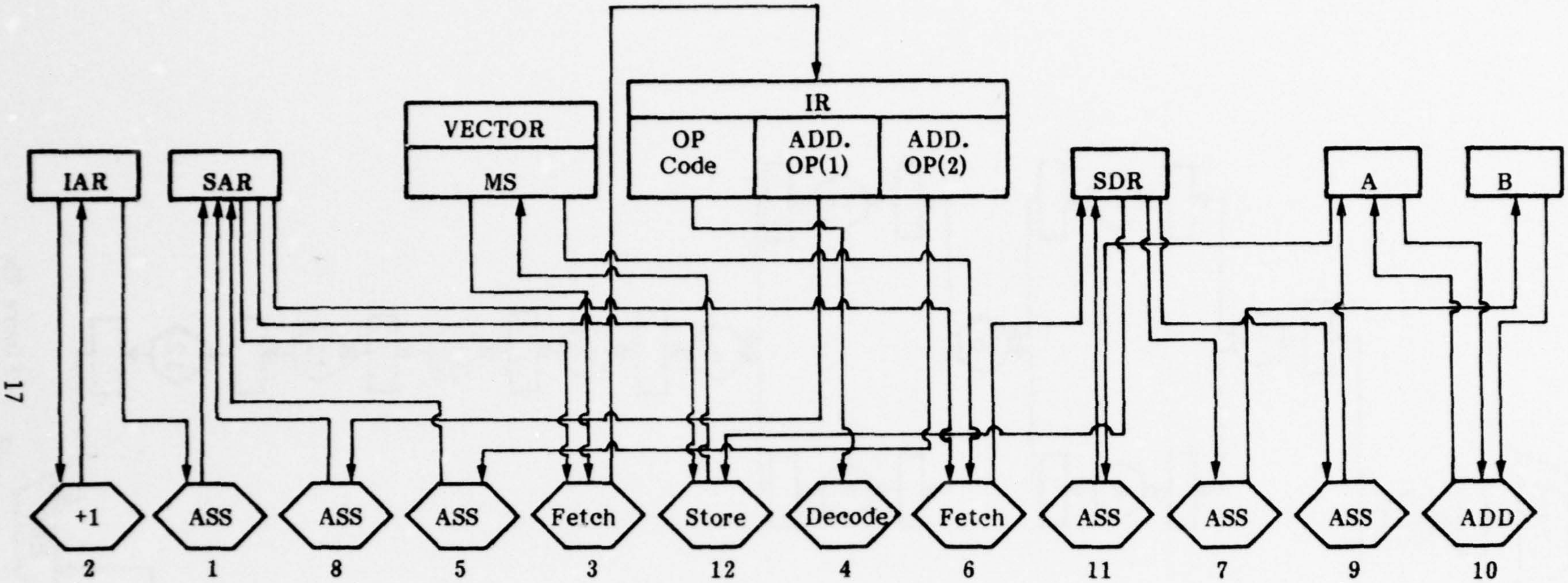


Figure 7. DFP Representation of Example Two



Data Graph For

Figure 8a. LOGOS Representation of Example Two

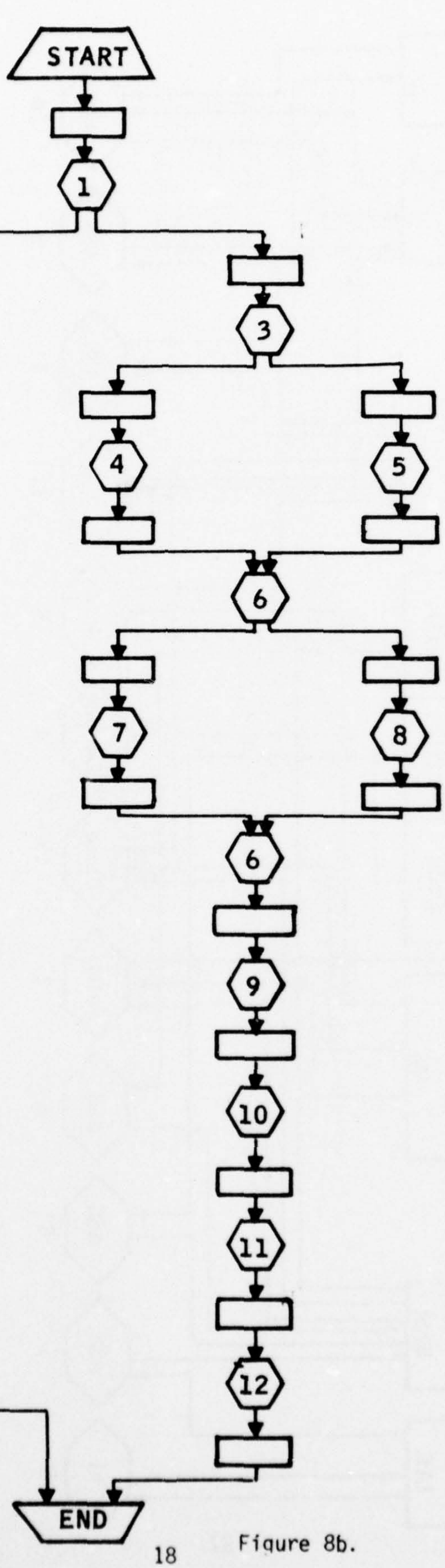


Figure 8b.

1) Control Signal Selection of Data Values--The mechanism in the graph by which control signals determine which data item is used from a data structure.

2) Data Steering of Control Sequences--The mechanism by which data values influence the flow of control in the system.

3) Deviation from Real Systems--The degree to which the structure and sequencing of the labelled graph model mirrors the structure and behavior of the actual hardware or software system being represented.

4) Exposure of Parallelism--The extent to which any parallelism or concurrency in the architecture being modeled is reflected in the structure and behavior of the model.

5) Representation of Data Instances--Some models directly represent data storage elements which can be referenced by "read" operations and can receive data with "write" operations. Other models create a new "read-only" data element following each data operation.

6) Determinancy--Determinancy is a labelled graph property relating to the certainty in the control sequence of a graph. Given an initial marking, a graph is determinant if its firing sequence of the transition is unchangeable.

7) Safeness--Safeness is a labelled graph property relating to the number of tokens in a place. A labelled graph is called safe if there can be at most a single token in each of its places at any time.

8) Representation of Synchronous Systems--Synchronous or asynchronous systems differ mostly in the way activation condition is specified; i.e., whether it is a signal coming from a central clock or a signal produced at the completion of the preceding or a combination of both. Labelled graphs are usually used to represent asynchronous systems or events, while it is also possible to represent synchronous systems or events by labelled graphs.

The single graph DFP models combined both data and control information in a single, somewhat complex graph. We observed the following features in these models:

1) Control Signal Selection of Data Values--T-gate, F-gate and merge actors permit the outcomes of deciders to decide the flow of values. If an operator has an *input control arc, this operator cannot be* enabled until the arc contains a control token.

2) Data Steering of Control Sequences--Decider elements accept data tokens as inputs and use these to output control tokens to initiate control sequences.

3) Deviation from Real Systems--An actor cannot fire if any one of its input arcs contains no token. This does not correspond completely with realistic hardware systems in which every memory cell contains information no matter if it is the desired information or a left-over from previous calculations. On the other hand, this model seems able to model software programs reasonably well by envisioning every token placement as a snapshot of the program conditions.

4) Exposure of Parallelism--From the examples illustrated, the computations represented by the data flow programs deviate from the original

algorithms. Hidden parallelism of the original algorithms are uncovered during the process of constructing data flow programs. As a consequence, this construction of a data flow program from a system description or an algorithm is not straightforward.

5) Destructive Read (Dynamic Memory Cell)--Since the firing of an actor or a link removes all the tokens on the input arcs, it can be envisioned as destructive read. Tokens to be used more than once are replicated by links.

6) Safeness--Safeness is enforced by the DFP firing rules

7) Determinancy--Also because of the firing rules, data flow programs are deterministic.

8) Representation of Synchronous Systems--Synchronous systems or events can be represented by adding a control token generator which has an output arc connecting to every operator. This control token generator generates control tokens as the clock of the simulated systems. It functions as a clock.

In LOGOS, control and data are separated and are represented by a control graph and a data graph, respectively. Although it may be redundant, sometimes the separation does increase clarity. Features observed in LOGOS are listed below:

1) Control Signal Selection of Data Values--Every d-operator (data operator) in a data graph is associated with at least one atomic operator in the corresponding control graph. When the atomic actor is activated, all of its associated data operators start to perform their data access and data transformation. When all the data operators complete their operations, the operation of the atomic actor is then completed. It is important to note that in the

original LOGOS [17]every d-operator is uniquely associated with only one atomic operator. This change is introduced to increase the clarity.

2) Data Steering of Control Sequences--PRED (predicate) operators in LOGOS are the counterpart of deciders in data flow programs. They are data dependent control branches whose data operator performs a test on its input d-cells (data cells). The outcome of the test decides which control branch will be taken.

3) Deviation from Real Systems--The deviation problem of data flow programs does not exist in LOGOS.

4) Exposure of Parallelism--Hidden parallelism need not be found to construct a LOGOS model. Note that it is possible to identify hidden parallelism after constructing a LOGOS model, if desired.

5) Determinancy--It is not necessary that a control graph be deterministic.

6) Safeness--Safeness is not necessarily observed in control graphs. In data graphs, operands are stored in data cells and it is non-destructive read. Thus token content does not play a role in data graphs.

7) Representation of Synchronous Systems--Synchronous systems or events can be represented by adding a token generator which has an output arc connecting to every operator of the control graph only. This token generator functions as a clock.

It is hard to judge objectively whether LOGOS or DFP is more suitable in general. After all, they were developed for different goals. We feel that the use of a static data cell (memory) for every variable and the non-restriction in the safeness of the firing rules of LOGOS are more pertinent and natural for modeling. On the other hand, others may consider a one graph approach to be more readable. We have constructed a new one graph model with the desirable features of LOGOS (static data cells and unrestricted firing rules).

The new model introduces sequence links for specifying the precedence relationships in a control sequence. If a transition (operation) has input sequence links, then it cannot be activated until there exists one or more token(s) in every one of these sequence links. If a transition has output sequence links, then, after the transition operation is completed and its outputs are stored in the corresponding data cells, a token is placed on every one of its output sequence links. Example One and Example Two are reconstructed using sequence links in Figures 9 and 10, respectively. Note that we can also add some operations on sequence links to specify complex control sequences.

For systems with simple control sequences, a one graph approach may increase the conciseness of the model, since it does not need two operators, one for control and the other for data, to specify the same event. But on the other hand, for systems with complex control sequences, the addition of sequence links and their operations may clutter the diagram and reduce the clarity. In general, we think the two graph approach is a reasonable one for modeling complicated systems. In some cases we may even need more specifications and descriptions of every individual graph to make it clear.

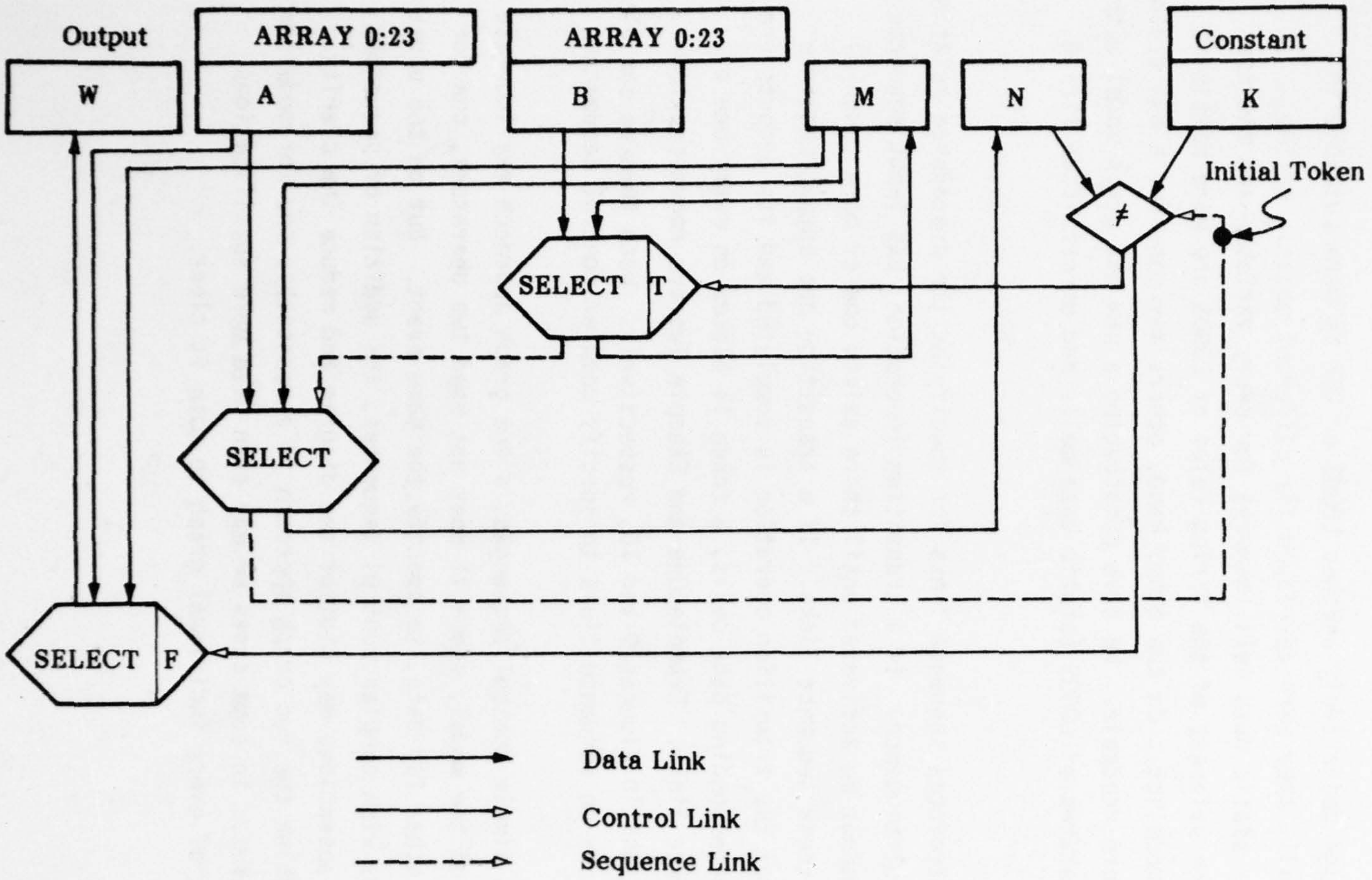


Figure 9. Alternate One Graph Model Representation of Example One

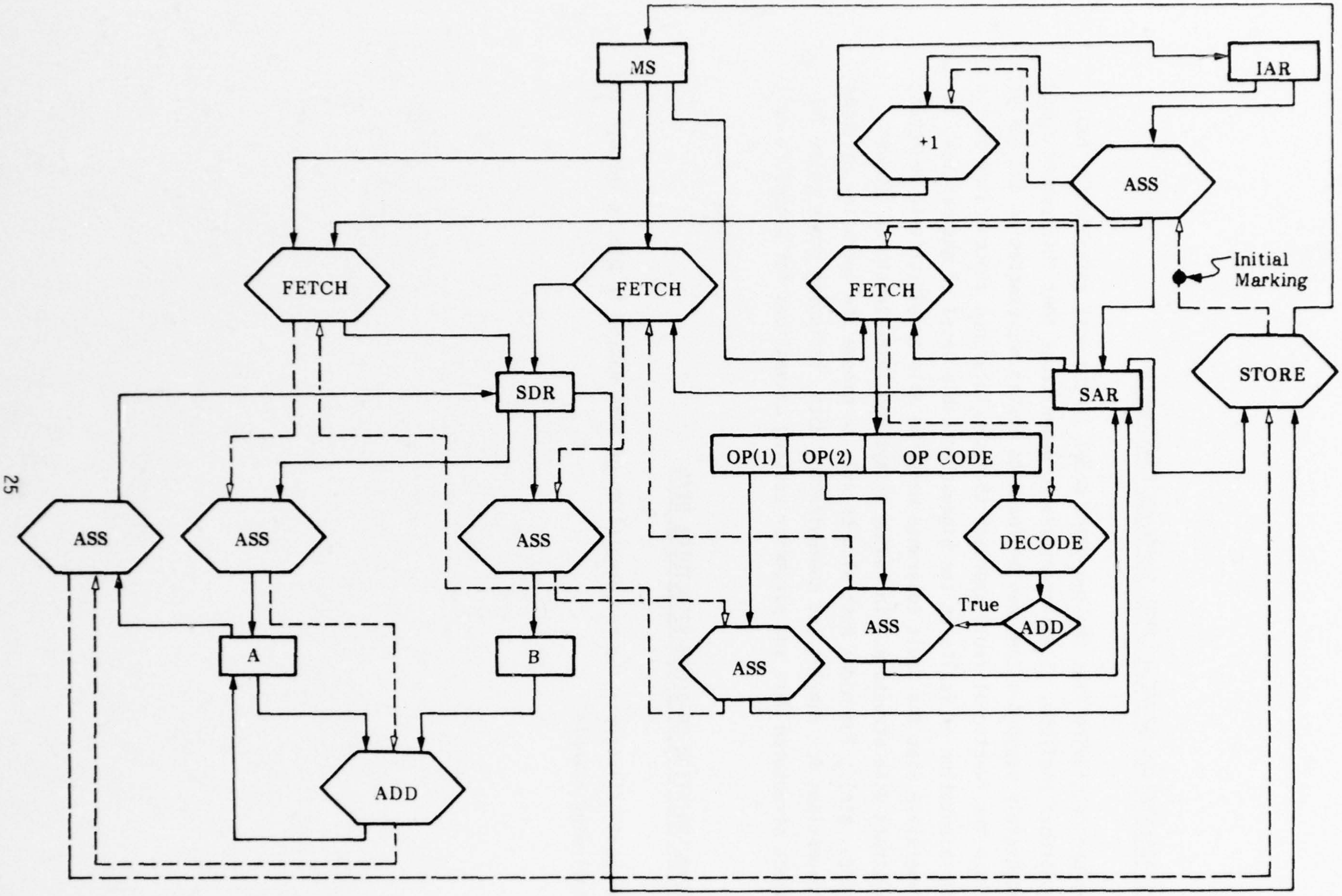


Figure 10. Alternate One Graph Model Representation of Example Two

REPRESENTATION OF HIGH LEVEL SOFTWARE CONSTRUCTS

Before concluding that the two graph model is the most suitable for fault tolerance analysis, it was necessary to demonstrate that this scheme could adequately support a top-down approach to system representation and analysis. Since the functional fault approach to fault tolerance concentrates on the representation of faults at the highest possible level of abstraction, it is imperative that the fault tolerance model be able to effectively represent abstract data structures, including arrays and list structures (queues, rings, etc). The model must also be able to expose the operation of common conventions for sharing and communicating data, including scope rules for block structured data and parameter passing conventions for procedure calls.

REPRESENTATION OF BLOCK-STRUCTURED DATA

We shall illustrate the representation of data nested in blocks through the following example:

```
PROGRAM A;
     INTEGER:  a, b;
         ●
         ●
         ●
        PROCEDURE B;
          REAL:  c, d;
              ●
              ●
              ●
            PROCEDURE C;
              INTEGER:  e, f;
                  ●
                  ●
                  ●
              END;
          END;
          ●
          ●
          ●
          PROCEDURE D;
           INTEGER:  g, h:
               ●
               ●
               ●
           END;
   ●
   ●
   ●
END.
```

The nested block structure of the above program is shown in Figure 11a.

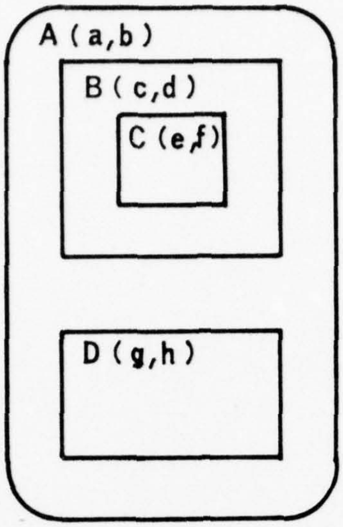


Figure 11a.

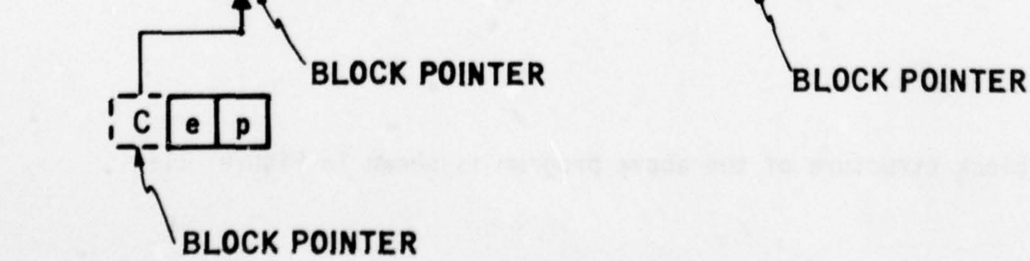


Figure 11b.

The data cells for this example are depicted in Figure 11b. For ease of illustration, the data cells of a block are grouped as a row--let us call it a data row. In most implementations, the individual data items would be stored contiguously in the form of an array or a stack. Each data row has a block pointer data cell associated with it which points to the immediate higher level block. Pointer data elements are essential for up-level addressing, but they are usually inaccessible and invisible to the user at the programmer level, so we represent block pointers as dotted-line data cells at the beginning of the data row with an arc to the higher level block. For further convenience, we label the block pointer for each data row with the name of the corresponding block.

Linked together by these block pointers, the data cells of a program are now structured as atree. When the control of a program is in a block, then the available data cells are the ones belonging to this block and to all higher level blocks. These cells are linked and can be traced via the block pointers starting from the data row of the current block to the root of the tree. For instance, if the control enters block B of the preceding example, then the available data cells are the ones of Block A and of block B. It is important to note that there is one and only one path from a node (data row) of a tree to its root [35].

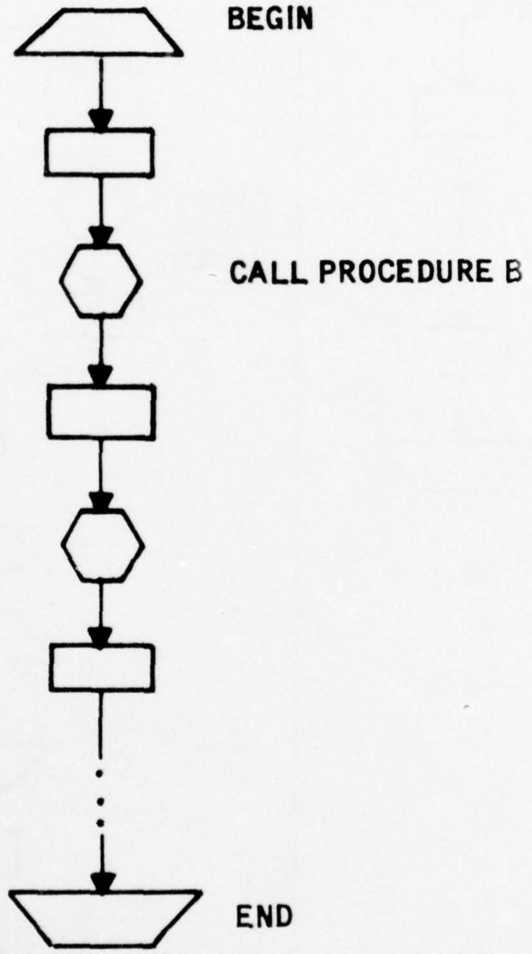
PROCEDURE CALLS

The transfer of control and data parameters to and from procedures that are shared by more than one user have always been a problem for graphical system models. In some cases, we may be able to treat the shared procedure as if it were an open subroutine by providing a separate copy of its data graph and control graph to each potential user. This process becomes difficult when a reasonably complex procedure is shared by many users because of the problem of including so much redundant information in the model. Also, we will often require a model in which a change in the structure of a commonly used procedure, possibly due to a fault, immediately affects all users of the procedure. In such cases, we want to use a single copy of the shared procedure control graph and link it to each of the potential users.

There are two kinds of procedures we must handle for procedure calls, serially reusable procedures and reentrant procedures. Serially reusable procedures (or programs) are the procedures which only allow one calling procedure to call it each time. This means that nobody else can execute this procedure until the current user, if existent, finishes his execution. The other kind of procedure is reentrant. A procedure is reentrant if, while it is being executed, anybody else may also begin executing it. (This is most useful in a multiprogramming or time-sharing environment, where several programs may call, say, a Sine routine which is reentrant. Only one copy of the routine need be kept in memory, no matter how many people are executing it.)

In general, most procedures are serially reusable. For simplicity, assume that the term procedure denotes a serially reusable procedure. This case is discussed first. Subsequently, with the use of colored tokens, we shall present a method for handling reentrant procedure calls.

The easiest way to explain the handling of procedure calls may be through the use of an example. Figure 12 shows the control graph of a program before executing the procedure-call operator. If a procedure-call operator is encountered it is assumed that a system routine (procedure) which handles procedure calls in invoked. In Figure 13 , a dynamic link is used to link the system routine with the procedure-call transition. Dynamic links are denoted by links with dotted lines. Dynamic links are the links which do not exist until a specific operator is executed and the linking information about the source and the destination, which has been stored in a corresponding data cell(s) is fetched to construct the link dynamically.



CONTROL GRAPH BEFORE EXECUTION OF A PROCEDURE CALL

Fiqure 12.

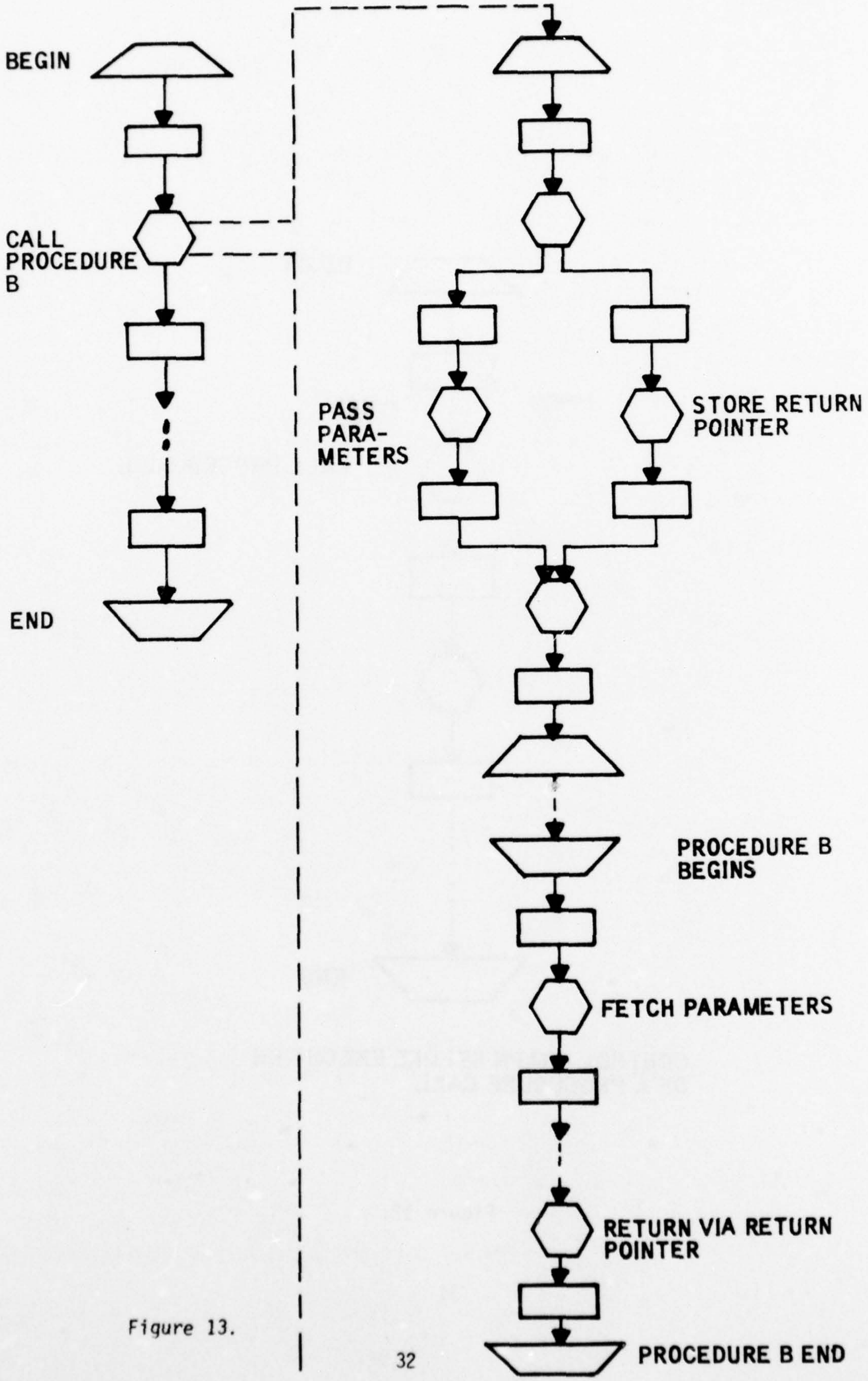

BEGIN
CALL PROCEDURE B
END
PASS PARAMETERS
STORE RETURN POINTER
PROCEDURE B BEGINS
FETCH PARAMETERS
RETURN VIA RETURN POINTER
PROCEDURE B END

Figure 13.

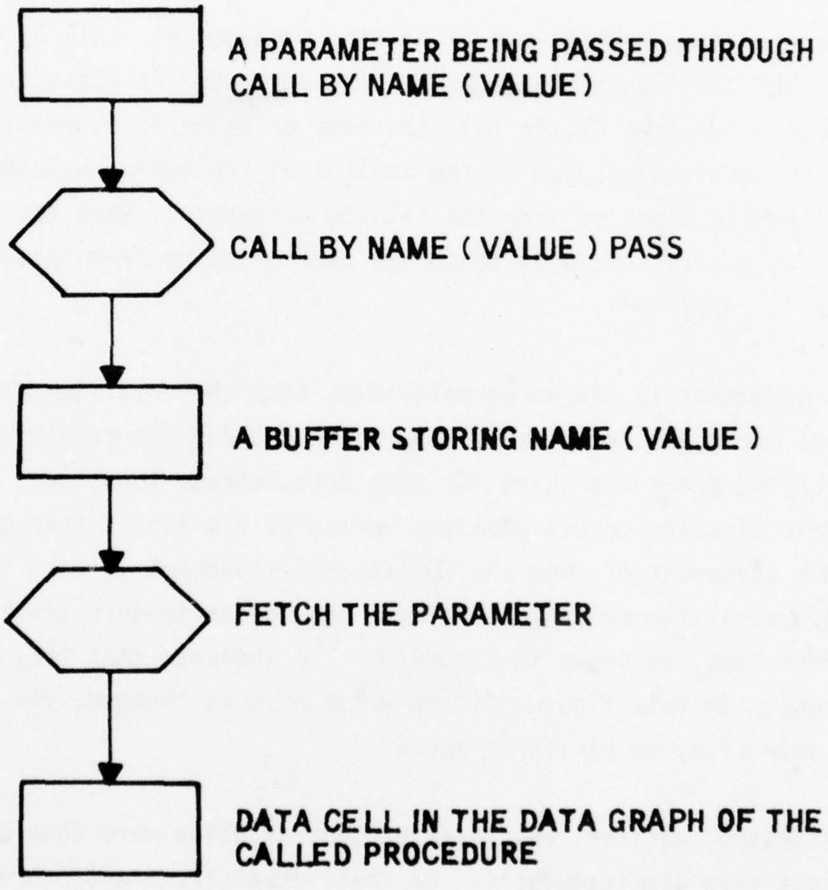
Before linking to the called procedure, B, via another dynamic link, this routine provides parameter passing and storage of the pointer pointing to the control graph of procedure B. If the system to be modeled has some other capabilities, for instance security checking, these operations are also performed by this routine.

There are three common ways of passing parameters: call by name, call by value and call by reference [34]. If a parameter is passed through call by name or value (see Figure 14), the name or value is stored in a buffer first. This buffer is allocated as the mailbox of the called procedure, to store the parameters passed from the calling procedure. When the called procedure acquires control, it will fetch the name or value from this buffer and store it in its data cell.

If a parameter is passed by reference, then the data item passed in as the formal parameter from the calling procedure and the receiving counterpart in the called procedure share the same data storage location. This same type of data aliasing occurs when two variables are linked through the FORTRAN COMMON statement or when the EQUIVALENCE statement is used in an assembly language environment. In this case the two parameters are linked by an address link, as shown in Figure 15 to indicate that they have the same address. In this figure, if the value of e is changed, the value of b follows. The pair (f,a) is similarly bound.

As discussed earlier, reentrant procedures allow more than one user to execute them simultaneously. We shall treat reentrant procedures in the same fashion as serially reusable procedures with the following extension:

1) In the control graph, identify different calls by different color tokens. A color can be assigned to a user according to his order entering the called procedure or preassigned priority.


A PARAMETER BEING PASSED THROUGH CALL BY NAME (VALUE)
CALL BY NAME (VALUE) PASS
A BUFFER STORING NAME (VALUE)
FETCH THE PARAMETER
DATA CELL IN THE DATA GRAPH OF THE CALLED PROCEDURE

DATA GRAPH PART FOR PARAMETER PASSING THROUGH- CALL BY NAME (VALUE)

Figure 14.

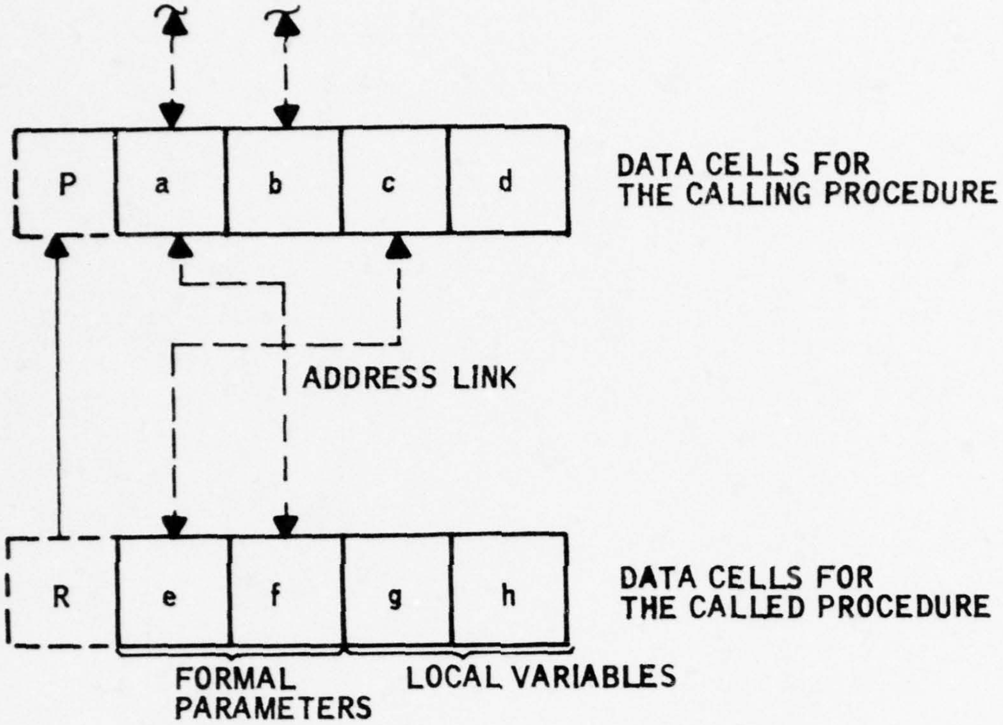

P
a
b
c
d
DATA CELLS FOR
THE CALLING PROCEDURE
ADDRESS LINK
R
e
f
g
h
DATA CELLS FOR
THE CALLED PROCEDURE
FORMAL
PARAMETERS
LOCAL VARIABLES

Figure 15.

2) Provide a copy of data cells which corresponds to each color of the tokens in the control graph.

SECTION III

DATA CONTAMINATION ANALYSIS

BACKGROUND

One of the main reasons for developing a representation for data in our fault tolerance model is to provide a vehicle for tracing the spread of data contamination in a system following a data fault. We are also interested in determining the location of data items that are ancestors of a particular critical data item in a system and hence may need extra monitoring or protection to guarantee the integrity of the critical data item. Both problems involve the tracing of the influence of individual data items for the time they are defined until their eventual use. Labelled graphs have not been extensively applied in this area, but conventional directed graphs have been used extensively for data flow analysis, principally for the optimization of code generated by compilers. We found the work of Allen and Cocke [9] in this area to be particularly relevant.

Directed graphs have been applied to fault tolerance analysis by a number of investigators. Ramamoorthy [2] pioneered the application of graph theory to fault tolerance. His work provided a systematic method to segment a digital system, to insert test points, and to open certain edges of the graph to make it loop-free for fault diagnosis. By considering that each unit did test the other units of a system and by implicitly assuming that each of the units had the same reliability, Preparata, Metze and Chien [3] and later Hakimi and Amin [4] used a graph model to derive the necessary and sufficient conditions for the system to be multiple-fault diagnosable with some connection assignments. With the same graph model, Russell and Kime and Barni, et al [5] modified some assumptions and extended the results in Preparata's paper [3] . Akers[7,8] pointed out that one of the most famous graph theory problems--namely the coloring problem--is closely related to the test generation problem of fault diagnosis, although the examples given are only loop-free graphs (combinational circuits) .

To exploit these analysis schemes based on conventional digraphs, we developed a procedure to convert a labeled graph system model into a directed graph model. The transformation procedure also allows us to determine if the original labeled graph will be unbounded, dead or deadlocked due to design errors. Subsequently, we shall use data flow analysis results originally developed for compiler optimization for the analysis of data contamination, which is the spread of erroneous data resulting from a fault or a set of faults. We also shall apply these data flow analysis results to study the other aspects of fault tolerance, i.e., tracing crucial input data and analyzing the effects of control sequence errors.

TRANSFORMATION OF PETRI NET-LIKE LABELED GRAPHS TO DIRECTED GRAPHS

In the following section we present a method for transforming a Petri net-like labeled graph to a directed graph which preserves the transition firing sequence of the original labeled graph. This method follows very closely Karp and Miller's [22] rooted tree approach for obtaining the reachable states of vector addition systems. A vector addition system represents the system state by recording the assignment of tokens to nodes in a vector having an element for every node. In these systems the firing of the next transition from any state is independent of the way by which the state was reached. As a result, we can very easily derive the reachable states. Since each transition is interpreted as an operation in our model, we prefer that a node in the transformed directed graph denote the firing of a transition. This can be done by performing another graph-to-graph transformation after obtaining a directed graph in which every node is a state in the vector addition system.

Before the discussion of our method, we briefly introduce vector addition systems.

Definition: An r-dimensional vector addition system V is a pair V = (s,W) where

(1) $s \in N^r$, $N = \{0, 1, \ldots\}$. This is the initial state.

(2) W is a finite set of r-dimensional integer vectors

$W = \{w_1, \; , \ldots w_k\}$, $w_i \in \{0, \pm 1, \pm 2, \ldots\}$

This is the set of firing rules.

The reachability set R(V) is the set of vectors given by:

$R\ (V) = \{X_i / X_i = s + w_{i1} + w_{i2} + \ldots. + w_{ik}\}$ where

$w_{ij} \in W$, $j = 1, 2, \ldots k$, and $s + w_{i1} + w_{i2} + \ldots + w_{ik} \geq 0$

The reachability set is the set of all points that can be reached by some path from s using vectors from W and which have no negative elements in their position vectors.

Example - As an example of a vector addition system, consider

V = (s,W)

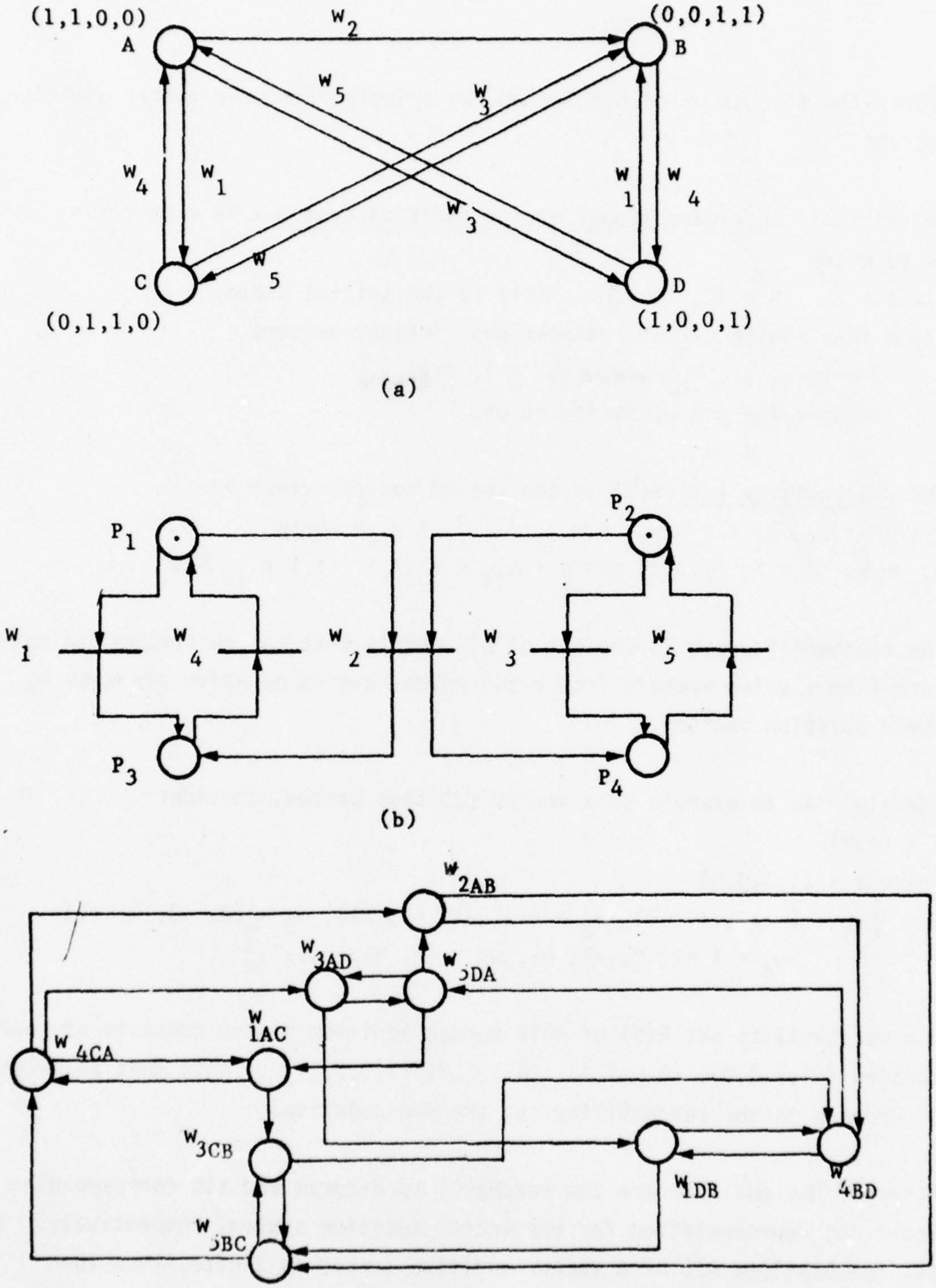
where s = (1,1,0,0)

$W = \{w_1 = (-1, 0, +1, 0), w_2 = (-1, -1, +1, +1), w_3 = (0, -1, 0, +1)$

$w_4 = (+1, 0, -1, 0), w_5 = (0, +1, 0, -1)\}$.

The reachability set R(V) of this vector addition system consists of four vectors $\{(1,1,0,0), (0,0,1,1), (0,1,1,0)\ (1,0,0,1)\}$. Note that all elements of vectors in the reachability set are non-negative.

Figures 16a and 16b are the reachability diagram and its corresponding Petri net representation for the vector addition system, respectively. If the reachability set of a vector addition system is finite, then the



(a)

(b)

Where W_{5BC} corresponds to the edge W_5 in Figure 4(a), from node B to node C, etc.

(c)

Figure 16.

reachability diagram is a finite machine, as in Figure 16a. On the other hand, if the system is unbounded, we need a method to detect it and stop the transformation. For this purpose, the following notation and definitions are introduced:

A vector addition system (Petri net) is called unbounded if it contains an infinite number of tokens; it is called dead if it is free of tokens; and it is called deadlocked if it contains tokens but cannot fire any transition. For example, Figure 17 shows a net which will be deadlocked if a vector v_i, $(x_1, x_2, \ldots x_r)$, can be reached from another vector v_j, $(y_1, y_2, \ldots y_r)$, and $x_k \geq y_k$, $1 \leq k \leq r$, then certainly at v_i we can repeat the process which has brought v_j to v_i. Thus, for the elements for which $x_k > y_k$, x_k can be increased monotonically to infinity. Therefore, this system (or Petri net) is unbounded. Also if $x_k = y_k$, $1 \leq k \leq r$, we can stop the process which generates other vectors (or states) from v_j.

The procedure for generating a reachability diagram of a given vector addition system is summarized in Algorithm 1. This algorithm can determine if a given vector addition system will be unbounded, dead, or deadlocked. If not, the algorithm generates a directed graph in which every node is a state in the sense of vector addition systems and every edge corresponds to a transition firing.

Algorithm 1:

Define a new state as a state which has not appeared yet in the previous generation of the reachability diagram.

Step 1: Is there any new state left? If no, STOP.

Step 2: Can this state, say state A, have any successor? If yes, GO TO step 3. If not, it is either a dead state or a deadlock. If this state is free of tokens then it is a dead state; otherwise, it is a deadlock. STOP.

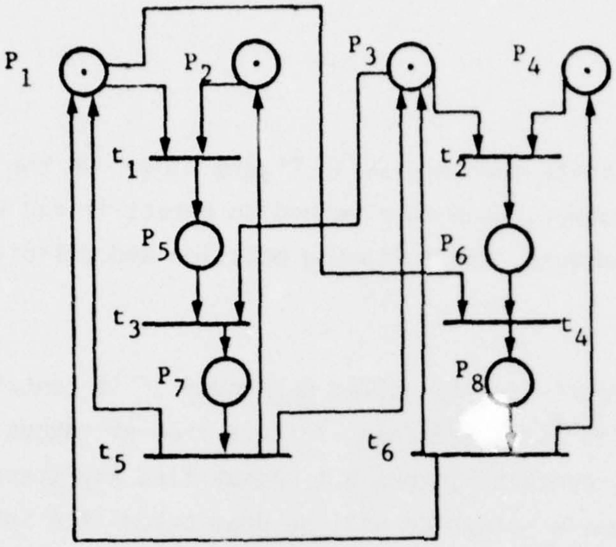


Figure 17.

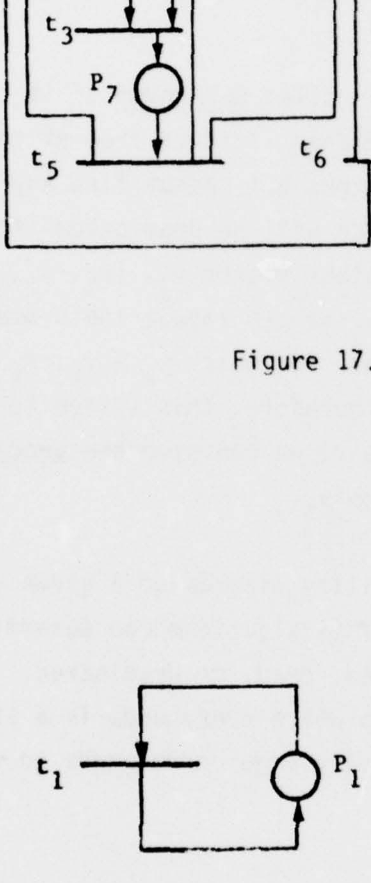


Figure 18.

Step 3: Determine whether there is any successor remaining, if no, GO TO step 1, if a successor remains, remove the first such succes~~or~~ (call it state B) from the rest of the remaining successors and determine if state B has appeared before. If not, state B is a new state; otherwise it is an old one. Trace the ancestors of state B to see whether it has a smaller ancestor, if so, it is unbounded, so STOP. Note that if state B is an old state, only state A and the ancestors of state A have to be traced, because the others have been traced already. GO TO step 3.

End Algorithm 1.

The process of transformation is not complete yet, however, because we want every transition in the labeled graph to correspond to a node in the directed graph, while in the diagram resulting from algorithm 1 it is an edge. This reachability diagram has to be transformed to another directed graph in which every node, say node a, corresponds to an edge, say edge a, in the diagram and there is an edge from node a to node b in the graph, if there is a node which edge a enters and which edge b exits in the reachability diagram. The algorithm for this portion of transformation can be found in [23] . In the previous example, the reachability diagram, Figure 16 a , is transformed to the directed graph, Figure 16 c , thus, completing the process of transforming a Petri net-like labeled graph to a directed graph while preserving the same firing sequence of transitions.

Not all the Petri nets can be represented as a vector addition system unless we add some modification to ordinary vector addition systems. For instance, there is no vector addition system counterpart to a Petri net that contains the net shown in Figure 18 . This modification is beyond the scope of this discussion.

The above transformation, transforms our labeled graph model, a Petri net-like graph, to a directed graph. Every node of the directed graph corresponds to a transition of the control graph, which, in turn, corresponds to a set of data information. In the transformation, the data information is transferable if desired. This data information can be used to trace the spread of data contamination.

DATA CONTAMINATION ANALYSIS

Given a directed graph in which each node performs a data operation, this section proposes an approach for tracing data contamination due to either faulty input data or incorrect data operations at some nodes. This approach utilizes some data flow analysis results which will be introduced first.

For compiler optimization, it is frequently desirable to know where data items are defined and where they are used. Some examples are "reaching definitions," "upwards exposed uses" [24], "live variables," "very busy variables," [25], etc. These problems, because they can be solved basically in the same manner, are called "global data flow analysis problems."

Allen and Cocke [9] use an "interval analysis" approach to solve these problems and Kildall [26] suggests an obvious method to handle them. It has been proven [24] that this obvious method is as good as the interval algorithm for solving all known global data flow problems. Also, a "node splitting" method must be used when graphs cannot be partitioned into intervals (irreducible graphs); the simple algorithm [24] proposed by Kildall still works on irreducible graphs. Kou [27] proposes a linked-list implementation scheme instead of the conventional bit-vector one for these problems and contrasts the computational complexities of difficult approaches. But, note that there are some deviations in definitions in [9] and [27]. Here, it is not intended to propose new algorithms or to compare their complexities for the data flow analysis problems; rather our concern is to show that some of these results can be utilized for fault tolerance.

Before discussion, we define the terms we are going to use. Most of these definitions follow the ones in [9].

A *data definition* is an expression or part of an expression which specifies a data item. A *data use* is an expression or part of an expression which references a data item without modifying it. Let DB_i denote the set of definitions defined at node i and UB_i denote the set of definitions used at node i. A *basic block* is a linear sequence of program instructions having one entry point (the first instruction executed) and one exit point (the last instruction executed). The nodes of the control flow graph can represent the basic blocks of a program. A *locally available* definition of node i, denoted as UB_i is the last definition of the data items in the basic block. A definition X in basic block n_i is said to *reach* basic block n_k if

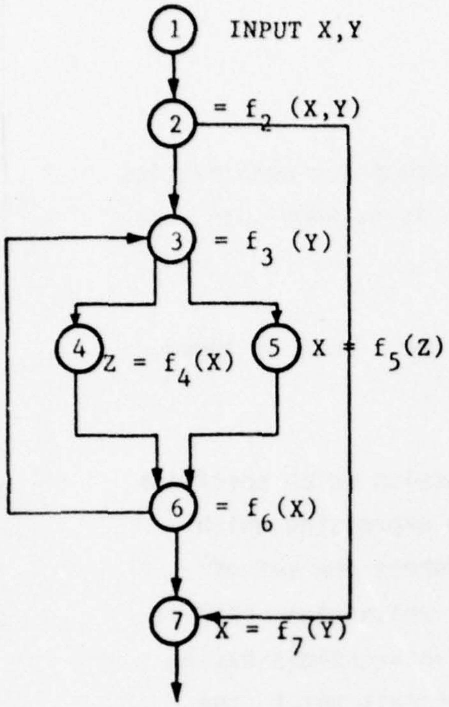
1. X is a locally available definition from n_i,

2. n_k is a successor of n_i and

3. There is at least one path from n_i to n_k which does not contain a node having a redefinition of the same data item; that is, X is preserved on some path from n_i to n_k.

The set of definitions reaching node i is called reaching definitions of node i and is denoted as R_i.

Figures 19a and 19b illustrate these definitions.



(a)

Node	DB_i	UB_i	R_i
1	X_1, Y_1	Ø	Ø
2	Ø	X, Y	$X_1 \quad Y_1$
3	Ø	Y	$X_1X_5Y_1Z_4$
4	Z_4	X	$X_1Y_5Y_1Z_4$
5	X_5	Z	$X_1X_5Y_1Z_4$
6	Ø	X	$X_1X_5Y_1Z_4$
7	X_7	Y	$X_1X_5Y_1Z_4$

(b)

Where the subindex of a variable indicates the node number in which the variable is related.

Figure 19.

Data contamination can result from input errors, system program malfunctions, corresponding hardware failures, etc. Now, consider the following problem: given a data flow analysis graph, how does contaminated data propagate.

One major difference between the data constant propagation studied extensively for compiler optimization and contaminated data propagation in faulty systems is that once a data item is contaminated, it may affect other data items, thereby propagating the contamination further in the system.

In discussing this propagation of contamination, we will use the following notation:

CID = contaminated variables

CID_i = with order contaminated variables. If i = o then it is the original contaminated input data.

Obviously, $(R_i \cap UB_i)$ is the set of input data used at node i. Take node 2 of Figure 19 a as an example.

$R_2 \cap UB_2 = (X_1, Y_1, Z_4) \cap (X, Y) = (X_1, Y_1)$. This tells us that at node 2, the variable X defined at node 1 and the variable Y defined at node 1 are used.

As an example of contaminated data propagation, consider the graph of Figure 19a again. Suppose that, due to a transient error in the transmission, the value of X received at node 4 is contaminated. It is also important to note that if f_4 (X) is miscoded due to a design error and the data item X available at node 4 is correct, the consequence is the same as in the previous case. We want to see which operations are affected and which variables are contaminated.

Obviously,

$CID_0 = \{X \text{ at node 4, transient error}\}$

$CID_1 = \{Z_4\}$

Note that neither X_1 nor X_5 is contaminated yet. But as Z_4 propagates and $UB_5 \cap R_5 = Z_4$

So the operation in node 5 is affected and

$CID_2 = \{DB_5\} = \{X_5\}$

Still, this contaminated X_5 propagates and

$UB_6 \cap R_6 = X_5$

Besides node 6, node 4 uses X_5 too, but it has been counted already. Now the operation in node 6 is affected but no variable is modified. This operation can be a test operation and whether the contaminated input X_5 causes the divergence of the control stream needs a sensitivity analysis.

Now, we obtain

$CID_3 = \{\emptyset\}$, so we do not need to proceed further.

Therefore, for the transient error of X at node 4 the affected operations are $\{4, 5, 6\}$ and the contaminated variables are $CID_1 \cup CID_2 = \{Z_4, X_5\}$.

The process for identifying affected operations and variables due to a set of contaminated data is summarized in the following algorithm. For simplicity, it is assumed that there is only one operation at every node. This assumption can be removed very easily be modifying the algorithm. Given a graph, DB_i, UB_i and R_i of every node i, and a set of contaminated data, this algorithm identified all the operations and variables by these contaminated data.

Algorithm 2:

Let A, B be a set of all the nodes and a null set, respectively.

Let CID_o be the set of originally contaminated data.

```
    Step 1   Initialization:  $n = o$; $CID_k = \emptyset$ for $k \geq 1$.

    Step 2   Scan through all the nodes in A, if it is $\emptyset$, GO TO step 3.

If node $i \subseteq A$ and $R_i \cap UB_i \subseteq CID_n$

then $A \leftarrow A$ - node i,    $B \leftarrow B \cup$ node i, and $CID_{n+1} \leftarrow CID_{n+1} \cup DB_i$

    Step 3   If $CID_{n+1} = \emptyset$, then GO TO step 4

             Otherwise

                  $n \leftarrow n + 1$;

                  GO TO step 2.

    Step 4   $CID = CID_o \cup CID_1 \ldots \cup CID_n$
```

The affected operations (their node numbers) are stored in B and the affected variables are stored in CID.

END algorithm 2.

In interpreting our graphical model, we consider a fault associated with a node as a data contamination error, while we consider faults due to missing edges or to extra edges as control sequence errors involving incorrect control paths. Generally, data contamination errors do not affect the sequencing of control until the contaminated data propagates to a node in which a conditional branching of control occurs. Contaminated data at a conditional branch node may cause faulty control sequences to be generated which use control paths that are not intended to be active under current system conditions. Whether this occurs depends on the error margin, i.e., the margin between normal data values and the threshold at which the branch decision changes, and a sensitivity analysis must be conducted to determine this error margin. In turn, a control sequence error with an incorrect path due to a missing edge or an extra edge can cause contaminated data, since the precedence relationship of executions may be changed. To analyze the consequences of a missing or an extra edge of a given graph and DB_i and UB_i, for each node i, a recalculation of R_i is needed. At a node, if X_1 is received and it should be X_g, the consequence is the same as with contaminated data discussed in the foregoing section. While if X_g should be received at a node and it does not, it is equivalent of use of an undefined variable. Again, it also can be as another form of data contamination.

In order to predict what kind of incorrect paths as well as contaminated data are created because of hardware failures, the model should have the following feature: A labeled graph model must contain information to identify the associated hardware with each data operation and each control operation; A directed graph model must contain information to identify the associated hardware with each node and each edge. Thus, given a likely failure node of an unreliable hardware component, we can trace the possible missing or extra edges. Through the analysis of the model, the consequence of failure can be derived.

Usually certain subsystems of a system are much more crucial than the rest in terms of system reliability, security or protection. Given a set of crucial operations, we may want to identify the corresponding hardware and inputs of these operations. The method for identifying all the related inputs of a given set of operations is presented below.

The approach for identifying all the crucial input data is similar to algorithm 2, but algorithm 2 traces the outputs and we now trace backwards to find the inputs. Let us use the graph shown in Figure 19a again as an example. Suppose that the operation performed by node 5 is crucial and we want to find all the inputs which affect this operation.

From Figure 19b.

$$R_5 \cap UB_5 = Z_4$$

So we know the input is Z and provided by node 4.

Now, $R_4 \cap UB_4 = X_1 X_5$

But node 5 has been scanned already, hence we only trace node 1.

Finally, $R_1 \cap UB_1 = \emptyset$

From this example we observe that as long as there is no control sequence error, only nodes 1 and 4 need function correctly to provide correct inputs to node 5.

SECTION IV

LABELED GRAPH MODELS WITH TIME

BACKGROUND

Our original set of functional fault classes included only asynchronous conditions in a system. Loss of control was represented by the loss of a token from the system, multiple instances of control were represented by the occurrence of spurious tokens in the system, and control deadlock was represented by Petri net structures with deadlock. Our committment to analyze the fault tolerance of real time control systems soon forced us to admit another type of functional fault, which we initially referred to as "tardiness of control". This refers to a condition in which a system sequences correctly and produces correct data values but does not complete its operations within the time constraints imposed by the real time environment. This may occur as the result of degradation of physical units in the system or as the result of the execution of unplanned, but legal sequences. A design error discovered in an actual operating system dramatically illustrates the latter situation. Engineers testing a dual-redundant real time control situation encountered a situation where the two channels would repeatedly fail to synchronize, although tests of both the system hardware and software showed them to be operating exactly as designed. The problem was finally traced to the system reaction to a transient fault sufficient to trigger a reinitialization sequence in only one of the two channels. The affected channel would complete its initialization sequence and, with the transient fault condition no longer present, would operate normally according to design. Unfortunately the design used a timer in each channel which would signal an error condition when the channel failed to complete its processing sequence in a specified time interval - an interval which was not large enough to allow the execution of both the initialization sequence and the subsequent normal processing sequence. The resulting timeout error caused another initialization se^uen^e in the next processing cycle, thus perpetuating the error condition. From a functional point of view, the errant channel was operating correctly, but not within system time constraints.

We also encountered problems with time in our efforts to model recovery schemes from loss of control faults. Although it was possible to develop schemes to detect this condition by monitoring the total token population in a subsystem, we had difficulties with models of detection schemes employing the familiar interval timer mechanism. Petri net models have "permissive" firing rules that allow an enabled transition to wait an arbitrarily long period of time before firing. Thus, it is impossible to create a convincing model of an interval timer that can be guaranteed to fire after a specified time interval without adding time parameters to the Petri net firing rules.

In a similar context, Merlin [29] studied two types of faults in Petri nets; loss of a token and gain of a token. Each net has a specified initial marking (or markings) and hence, a set of markings which are reachable from the initial marking. For any specified place, the "fault" would transform the markings of the Petri net by the removal of a token from the place (if it had one) or by the addition of a token to the place. He then defined a net to be recoverable from the fault if, in a finite number of transition firings, the net would return to a marking reachable from the initial marking in the normal net. On the other hand, if there is some infinite sequence of transition firings which leave the net in markings not reachable from the initial marking, then the net does not recover from the fault.

This approach could be applied to other types of "faults" which are, in fact, a transform of the net's markings. A fault is detectable if it transforms the "legal" markings of the net into "illegal" markings, i.e., markings not reachable from the initial marking. Recoverability would be defined the same as for loss of token or gain of token.

The concept of detectability could be important in itself. If a system is to be designed so that a certain class of faults are detectable, in the classical sense, then the design could be approached in a two step process. First, a Petri net must be specified for which each fault in the class is detectable as defined above. Secondly, the net must be specified so that all illegal markings cause the occurrence of one or more recognizable output events.

These ideas apply directly to some of the functional type faults. For example, multiple instances of control and loss of control are faults which lead to illegal markings and hence, by definition, are detectable. The multiple instances of control would be a fault, from which the system may or may not recover; however, by its definition, the system cannot recover from loss of control. Other classes of control faults, such as diversion of control and skew of control, may or may not be detectable for the given Petri net.

Merlin has shown that if a Petri net without time associated with transitions, is recoverable from loss of token faults, then the net has a "degraded" operation after the fault. It is "degraded" operation in the sense that not all markings which could occur before the fault can occur after the fault even though the net remains in legal markings. Time was introduced into the net to retain recoverability without the "degraded" operation. For example, his investigations indicate that there do not exist practical recoverable asynchronous communication protocols unless the execution time of events are known a priori.

REPRESENTATIONS FOR TIME IN A LABELED GRAPH MODEL

Labeled graph models are generally interpreted so that transitions denote events or sets of operations. Thus it is natural to link time parameters with the firing of transitions. There are several alternatives for specifying the firing time of a transition. These include:

- Specification of a fixed time interval for the transition firing
- Specification of upper and lower bounds for the transition firing time interval
- Specification of a probability density function describing the distribution of transition firing time interval values
- An enumeration of a set of possible transition firing times
- The specification of some "absolute" time before which the transition must fire, if enabled.

The last alternative requires the establishment of a system time reference and becomes a problem when a transition is in a loop. The first alternative is difficult to implement in a model that hierarchically models details of lower level system operations inside high level operators. In such a system, a high level transition may be expanded into a lower level graph that has more than one path from input to output. If two of these paths have different traversal times, then we must at least enumerate these alternatives for the higher level transition. The problem of enumeration becomes increasingly difficult in systems with a number of hierarchical levels.

The bounds approach and the probabilistic approach are the more practical alternatives, although probability distribution functions for multi-level systems with several alternative lower level paths will often be difficult to obtain. The bounds approach has the advantage that we only need know the best and worst case times of sequence at lower levels.

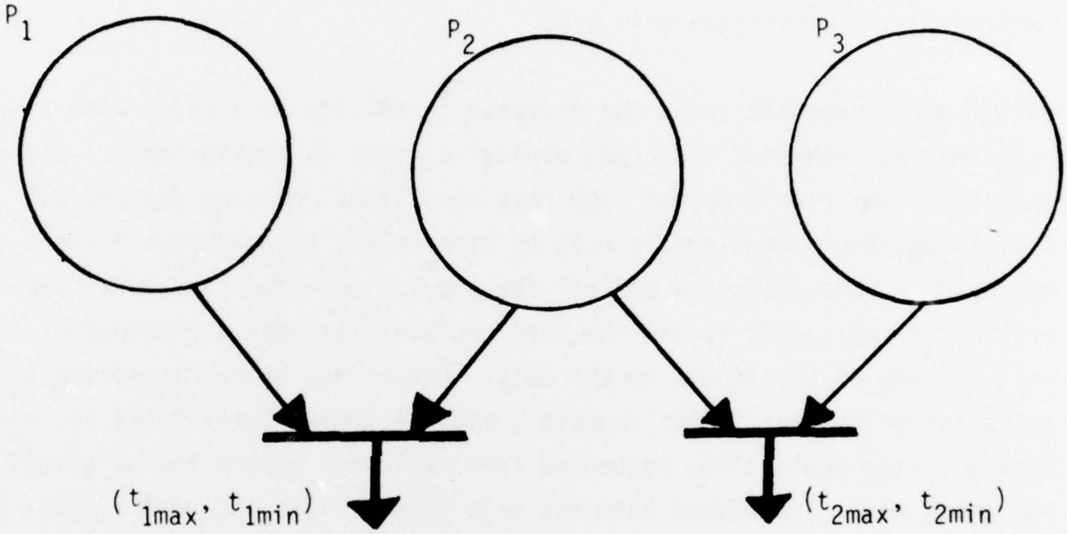
We follow Merlin's approach and assign two real numbers, $t_{i\text{-}min}$ and $t_{i\text{-}max}$, to a transition, t_i. We then require that a transition can only fire if it has been enabled for $t_{i\text{-}min}$ seconds, and that the transition must fire if it has been enabled for $t_{i\text{-}max}$ seconds. If $t_{i\text{-}min} + t_{i\text{-}max}$, then the transition always fires exactly $t_{i\text{-}min}$ seconds after it is enabled.

If all transition firing times are fixed with $t_{i\text{-}min} = t_{i\text{-}max}$, then, for finite nets, we can use the results of Ramchandandi [28] to determine firing schedules and computation rates. The process rapidly becomes complicated when $t_{i\text{-}min} \neq t_{i\text{-}max}$, and simulation may be necessary to characterize the operation of these systems.

EFFECTS OF ADDING TIME TO THE MODEL

The addition of time constraints to the model increases the representation power of the model. The addition of time parameters to a Petri net model without time constraints removes certain firing sequences from the set of allowable firing sequences. The sequences removed may be sequences leading to deadlocks, in which case a previously K-live net may become immortal. The interesting situations occur when time parameters are assigned to two transitions sharing a single place in a conflict situation, as in Figure 20 .

Clearly, if $t_{1min} > t_{2max}$, and the supply of tokens to place P_3 is replenished within $t_{1min} - t_{2max}$ seconds, transition t_1 will never fire. The replenishment rate for P_3 is a function of the number of tokens in the system and the firing times of the transitions in circuits of the graph that include P_3. It appears that, in nets with finite token populations, the ratios of transition firing



Conflict in timed net

Figure 20.

times may be adjusted to obtain the equivalent of a Petri net with priority between transitions sharing places. If this is true, then we may add time to simple Petri nets and obtain the representation power of a Petri net with negation if the nets involved all have finite token populations. We are continuing to investigate this area.

On the more pragmatic side, the increase in ability of a model with time to represent the behavior of actual digital systems is considerable. Although they sometimes risk problems from race conditions, present day digital systems designers continually rely on time delays to guarantee correct system operation. Some designers exploit the cumulative effects of gate delays in an arithmetic/logic unit to use the same register for both the output of the unit and one of the inputs to the unit. Others use delays in wiring to establish priorities in bus requests, etc. A conventional Petri net model of such a system would allow sequences that could not happen in the actual system. The timed model introduced here not only behaves like the real system, but also explicitly shows how the designer is using time to achieve his objectives.

The objective of this long-term project is modeling and analyzing fault tolerance aspects of real-time systems. In these systems, to satisfy time requirements is crucial for the success of the systems. Therefore, the transition firing times of the labeled graph model are specified for handling time-related faults such as race conditions and tardiness of control. In this section, we enumerate all the alternatives of specifying times of the labeled graph model, and observe the effect of adding time to the model. However, this is only a start of an urgently needed but hard area. We strongly suggest that further research should be conducted especially in analyzing the effects caused by the addition of time in transition firing sequences.

REFERENCES

[1] B. W. Boehm, "Software and Its Impact: A Quantitative Assessment," Datamation, pp. 48-59, May 1973.

[2] C. V. Ramamoorthy, "A Structural Theory of Machine Diagnosis," AFIPS Proc. SJCC, Vol. 39, 1967.

[3] F. P. Preparata, G. Metze, and R. T. Chien, "On the Connection Assignment Problem of Diagnosable Systems," IEEETC, Vol., EC-16, No. 6, December 1967.

[4] S. L. Hakimi and A. T. Amin, "Characterization of Connection Assignment of Diagnosable Systems," IEEETC, January 1974.

[5] J. Russell and C. Kine, "System Fault Diagnosis: Closure and Diagnosability with Repair," IEEETC, Vol. C-24, No. 11, November 1975.

[6] F. Barni, F. Grandoni, and P. Maestrini, "A Theory of Diagnosability of Digital Systems," IEEETC, Vol. C-25, No. 6, pp. 585-593, June 1976.

[7] S. B. Akers, Jr., "A Logic System for Fault Test Generation," IEEETC, Vol. C-25, No. 6, pp. 620-630, June 1976.

[8] S. B. Akers, Jr., "Fault Diganosis as a Graph Coloring Problem," IEEETC, Vol. C-23, No. 7, pp. 706-712, July 1974.

[9] F. E. Allen and J. Cocke, "A Program Data Flow Analysis Procedure," CACM, Vol. 19, No. 3, pp. 137-146, March 1976.

[10] C. A. Petri, "Kommunikation Mit Automation," Sohilften des Rein/sch-West and det Universitat Bonn, No. 2, Bonn 1962, Translated by C. F. Greene, Applied Data Research, Ltd., Technical Report No. RADC-TR-65-371.

[11] M. Hack, "Petri Net Languages," Technical Report 159, Laboratory for Computer Science, MIT, March 1976.

[12] J. D. Noe and G. J. Nutt, "Macro E-Nets for Representation of Parallel Systems," IEEETC, Vol. C-22, No. 8, August 1973.

[13] J. D. Noe, "Pro-Nets: For Modeling Processes and Processors," University of Washington, Tech. Report 75-07-15, Revised: 15 July 1975.

[14] A. W. Hold and F. Commoner, "Events and Conditions: An Approach to the Description and Analysis of Dynamic Systems," Third Semi-Annual

Technical Report, Part II, for the Project "Research in Machine-Independent Software Programming," Applied Data Research, Inc., April 1970.

[15] M. Jagayeri, "A Programmed Introduction for Users of a Computer Aided Design System," Report No. 1130, Case Western University, January 1974.

[16] C. W. Rose and M. Albarran, "Modeling and Design Description of Hierarchical Hardware/Software Systems," Proceedings of 12th Design Automation Conference, 1975.

[17] C. W. Rose, F. T. Bradshaw and S. W. Katzke, "The LOGOS Representation System," Digest Comcon 1972, September 1972.

[18] D. R. Slutz, "The Flow Graph Schemata Model of Parallel Computation," Project MAC, TR-53, Ph. D. Thesis, September 1968.

[19] J. B. Dennis, "First Version of a Data Flow Procedure Language," Project MAC, Group Memo 93-1, August 1974.

[20] J. B. Dennis and J. B. Fosseen, "Introduction to Data Flow Schemas," Project MAC, Group Memo 81-1, September 1973.

[21] A. Pritaker, B. Alan, and W. W. Happ, "GERT: Graphical Evaluation and Review Technique - Part 1 Fundamentals," The Journal of Industrial Eng., Vol. XVIII, No. 5, 1966.

[22] R. M. Karp and R. E. Miller, "Parallel Program Schemata," Journal of Computer and System Sciences, 1969.

[23] N. J. A. Sloane, "On Finding the Paths Through a Network," The Bell System Technical Journal, Vol. 41, No. 2, February 1972.

[24] M. S. Hachet and J. D. Ullman, "Analysis of a Simple Algorithm for Global Flow Problems," Conf. Record, ACM Symp. on Principles of Programming Languages, Boston, Mass., pp. 207-217, October 1973.

[25] M. Schaefer, "A Mathematical Theory of Global Program Optimization," Prentice-Hal, Englewood Cliffs, N. J., 1973.

[26] G. A. Kildall, "A Unified Approach to Global Program Optimization," Conf. Record, ACM Symp. on Principles of Programming Languages, Boston, Mass., pp. 184-206, 1973.

[27] L. T. Kou, "On Live-Dead Analysis for Global Data Flow Problems," IBM Research Rep. RC 5278, Thomas T. Watson Research Center, Yorktown Heights, N. Y. 1975.

[28] C. Ramchandani, "Analysis of Asynchronous Concurrent Systems by Petri Nets," MAC TR-120, Ph.D. Thesis, MIT, February 1974.

[29] P. M. Merlin, "A Study of the Recoverability of Computer Systems," Ph.D. Dissertation, University of California, Irvine, California, 1974.

[30] J. B. Dennis and D. P. Misunas, "The Design of a Highly Parallel Computer for Signal Processing Applications," Computation Structures Group Memo 101, Project MAC, MIT, august 1974.

[31] C. W. Rose, "A System of Representation for General Purpose Digital Computer Systems," Jennings Computing Center, Report No. 1113, Case Western Reserve University, Cleveland, Ohio, August 1970.

[32] D. Knuth, "Structured Programming with GO TO Statements", Computing Surveys, Vol. 6, No. 4, pp. 261-302, 1974.

[33] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, "Structured Programming", London, Academic Press, 1972.

[34] P. Wagner, "Programming Languages Information Structures and Machine Organization", McGraw Hill Series in Computer Science, 1968.

[35] F. Harary, "Graph Theory", Addison Wesley, 1968.

[36] L. A. Jack, W. L. Heimerdinger, and M. D. Johnson, "Theory of Fault Tolerance 1974-75 Annual Report," Sponsored by ONR under contract N00014-75-C-0011, September 1975.

18 19 REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER
AFOSR-TR-77-1255

2. GOV'T ACCESSION NUMBER

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (AND SUBTITLE)
6 A GRAPH THEORETIC APPROACH TO FAULT TOLERANT COMPUTING.

5. TYPE OF REPORT/PERIOD COVERED
9 Final Report.

6. PERFORMING ORG. REPORT NUMBER
14 FØ357

7. AUTHOR(S)
10 W. L. Heimerdinger [redacted] Y. W. Han

8. CONTRACT OR GRANT NUMBER(S)
15 F44620-75-C-ØØ53

9. PERFORMING ORGANIZATIONS NAME/ADDRESS
Honeywell Systems & Research Center
2600 Ridgway Parkway N.E.
Minneapolis, Minnesota 55413

10. PROGRAM ELEMENT PROJECT, TASK AREA & WORK UNIT NUMBERS
16 61102F
2304/A6
17 A6

11. CONTROLLING OFFICE NAME/ADDRESS
Air Force Office of Scientific Research (NM)
Building 410, Bolling AFB
Washington, D. C. 20332

12. REPORT DATE
11 12 Sept[redacted] 1977

13. NUMBER OF PAGES
61

14. MONITORING AGENCY NAME/ADDRESS (IF DIFFERENT FROM CONT. OFF.)
12 67p.

15. SECURITY CLASSIFICATION (OF THIS REPORT)
UNCLASSIFIED

15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (OF THIS REPORT)
Approved for public release--distribution unlimited.

17. DISTRIBUTION STATEMENT (OF THE ABSTRACT ENTERED IN BLOCK 20, IF DIFFERENT FROM REPORT)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (CONTINUE ON REVERSE SIDE IF NECESSARY AND IDENTIFY BY BLOCK NUMBER)

Petri Net
Fault Models
Functional Fault Classes
Environmental Errors
Siphons
Directed Graph Theory
Fault Tolerance
Physical Fault
Graph Properties
Invariants
Labelled Graphs
Functional Fault Models
Design Errors
Traps
Liveness

20. ABSTRACT (CONTINUE ON REVERSE SIDE IF NECESSARY AND IDENTIFY BY BLOCK NUMBER)

This report documents the activities in the second year of a two year investigation of a graph theoretic approach to fault tolerance for the Air Force Office of Scientific Research. This is part of a continuing effort also sponsored by the Office of Naval Research and by Honeywell, Inc. to develop a unified approach to the analysis of fault tolerant digital systems based on graph theory. Earlier efforts have examined existing graphical models and found a number of them to be suitable for fault tolerance modeling. Two models, Petri Nets and LOGOS were found to be particularly suitable. A subsequent effort examined available results in Petri net theory for properties and relationships applicable to fault

402 349

→ next page LB

Abstract - (Continued)

tolerance phenomena.

The effort documented here focuses on the incorporation of data aspects of the system in the model and on an explicit representation of time in the model.

This year's study results have convinced us that a two-graph labeled graph model that associates two time parameters with each transition or operation is a feasible and effective method of representing a fault tolerant digital system for analysis purposes. ~~We have not specified~~ the exact syntax of a single model* in this effort, but ~~we feel~~ such a definition can be made using these results in a straightforward way. Additional work is needed to identify data attributes critical to fault tolerance and to include them in the model.

* is not specified